REPLICATION

Nelson Onyibe and Genevieve Patterson

CS227

Monday March 5, 2012

A NEW APPROACH TO DEVELOPING AND IMPLEMENTING EAGER DATABASE REPLICATION PROTOCOLS

BETTINA KEMME AND GUSTAVO ALONSO

GOALS OF THIS PAPER

- Presents alternative to centralized approaches
  - These eliminate some advantages of replication
- Authors approach uses group communication primitives and relaxes isolation guarantees
- Authors present a form of compromise between Eager and Lazy replicaiton

COMPROMISE

- Desirable behaviors:
  - Correctness (ideal solution: eager replication)
  - Fault-tolerance (ideal solution: lazy replication)
- Authors wanted
  - More flexible than ensuring serializability
  - But with high correctness
- Proposed solution
  - Different levels of isolation of grouped, concurrently executed reads/writes
- Claim: their approach maintains data consistency

OUTLINE OF THE AUTHORS' PROTOCOL

- Basic steps in the authors' alternative implementation of eager replication
  - Perform transaction locally
  - Batch write operations
  - At transaction commit time deploy write sets to copies using TO multicast
    - This is similar to the 'push strategy' for lazy replication + ensured serial write operations
  - At reception time copies (and local site) check for conflicts
  - Because of TO multicast, conflict transactions are serialized
    - No need for 2-phase-commit
- Major Contributions: use of group communication, different levels of isolation, optimized fault-tolerance by use of TO broadcast

EXISTING TECHNOLOGY
(AT TIME OF PUBLICATION)

Table I. Classification of Replication Mechanisms

where \ when	Eager	Lazy
Primary Copy	Early Solutions in Ingres	Sybase/IBM/Oracle Placement Strat.
	Serialization-Graph based	
Update Everywhere	ROWA/ROWAA Quorum based Oracle Synchr.Repl.	Oracle Advanced Repl. Weak Consistency Strat.

- Where to update?
  - Primary Copy – simplifies concurrency but creates bottleneck
  - Update Everywhere – copies must be coordinated
- When to update?
  - Eager – detect conflict before propagation, ensures consistency
  - Lazy – propagate changes after commit, ensures maximum performance

EXISTING TECHNOLOGY
(AT TIME OF PUBLICATION) CONT'D

- Timeline of replication solutions:
  - Primary copy, eager replication
  - Update everywhere
    - Quorums (example of isolation)
    - Epidemic protocols
  - Lazy replication
    - Favored commercially
    - Push strategy – updates propagated directly after transaction commit
    - Pull strategy – update occurs only on client request
    - Both strategies can be used with primary copy or update everywhere
    - Trade Off: update everywhere + lazy replication = reconciliation complexity
- How should the best solution be selected based on the demands of the database? (not clearly discussed)

COMBINING EAGER AND LAZY TECHNIQUES

- The authors reference a previous system that used
  - Distributed locking
  - Global serialization graphs
  - Propagation after commit
- to combine advantages of Eager and Lazy protocols
- This previous attempt at combination used a primary copy implementation, and was scalability-limited

IMPROVING EAGER REPLICATION

- Authors combine correctness of eager with performance of lazy by using these techniques
  - Reducing Message Overhead
    - Bundle operations (i.e. ‘write sets’) as in optimistic schemes
  - Eliminating Deadlocks
    - Pre-order transactions – total-order broadcast
  - Optimizations Using Different Levels of Isolation
    - The more levels of isolation of operations, the closer this system gets to eager replication
    - More understandable by developers
  - Optimizations Using Different Levels of Fault-Tolerance
    - Correctness proportional to network reliability

COMPARISON OF DATABASE REPLICATION TECHNIQUE BASED ON TOTAL ORDER BROADCAST

MATTHIAS WIESMANN AND ANDRE SCHIPER

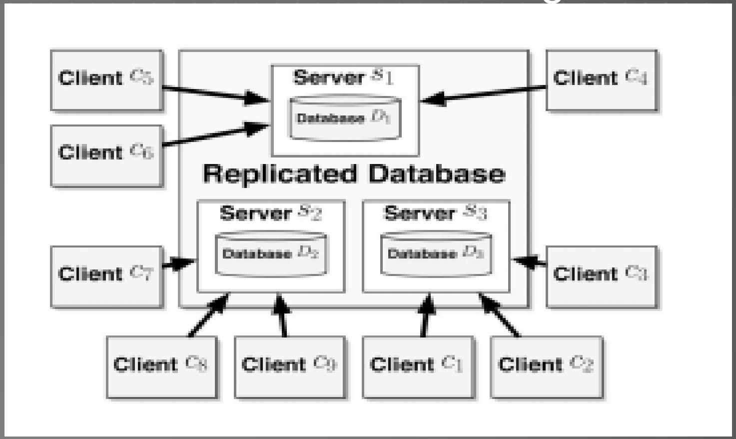
INTRO

- Techniques based on group communication typically rely on a primitive called TOTAL ORDER BROADCAST
  - Ensures that messages are delivered reliably and in the same order on all replicas
- Carried out
  - Eagerly
    - Within the boundaries of a transaction
    - Replicas are identical all the time
    - Conflicts detection before commit
    - Increased response time
  - Lazily
    - Delayed updates
    - Conflicts could creep in
    - There may exist inconsistencies among replicas

MODEL

- Server , S = {S_1, S_2, …, S_n}
- Each server S_i contains a full database, D
- One-copy serializability (All copies of D are kept synchronized at all times)
- Server S_i hosts a local transaction manager
- The local transaction manager ensures ACID properties of local transactions
- The local transaction manager TMi executes transactions that updates Database, Di
- Client , C = {C_1, C_2, …, C_m}
- The server that a client Ci contacts to execute a transaction, t is a delegate server for t
- In primary copy replication, only one server can act as a delegate server

Database Replication Model

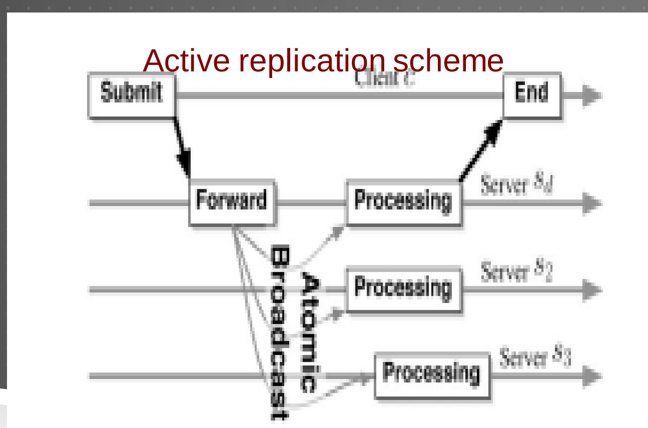


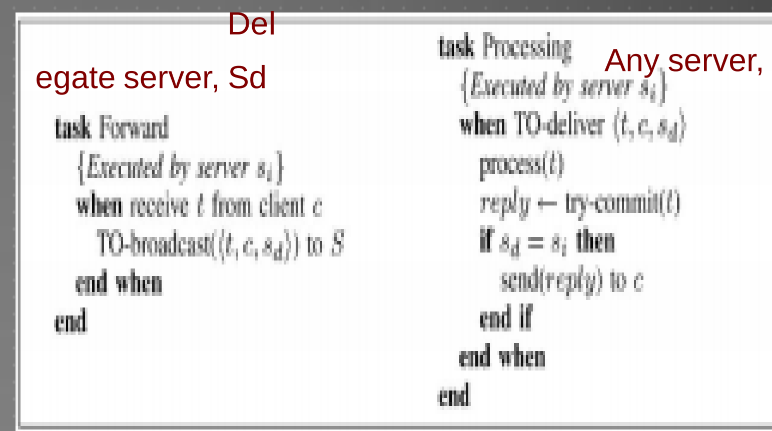
REPLICATION TECHNIQUES

- Group Communication Based Replication
  - Active Replication
  - Certification Based Replication
  - Weak Voting Replication
- Non Group Communication Based Replication (Just for Comparisons)
  - Lazy Replication
  - Primary Copy Replication

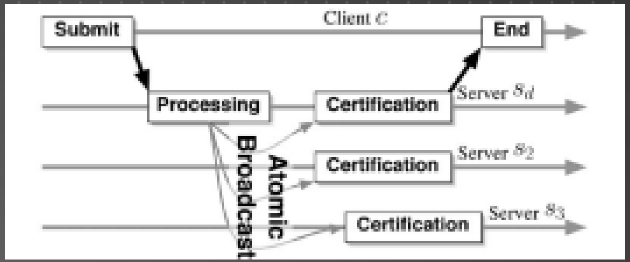
ACTIVE REPLICATION

- Client, C contacts server, S_d to execute transaction, t
- Server, S_d puts transaction, t into a messages, m
- Server, S_d broadcasts m atomically to all servers
- On receiving m, server, S_r serializes t
- Server, S_r processes t
- If any server, S_i aborts, all servers abort





CERTIFICATION BASED REPLICATION



- Client, C sends a transaction, t to server,
- S_d executes t but delays write operations
- When commit time is reached, the delayed write set in t is put into a Message, m and broadcasted to all servers using total order
- Upon delivering m, each server, S_i executes a deterministic certification phase that decides if t can be committed or not

Any Server Si

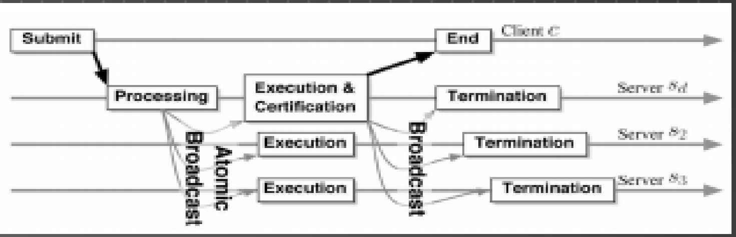
```
task Certification
  {Executed by server s_i}
  when TO-deliver ⟨readSet_t, writeSet_t, c, s_d⟩
    status ← certify(readSet_t, writeSet_t)
    if status = commit then
      if s_d ≠ s_i then
        execute writeOperations_t
      end if
      commit(t)
      if s_d = s_i then
        send(committed) to c
      end if
    else
      abort(t)
      if s_d = s_i then
        send(aborted) to c
      end if
    end if
  end when
end
```

Delegate Server, Sd

```
task Processing
  {Executed by delegate server s_d}
  when receive trans. t from client c
    execute trans. t
    if aborted(t) then
      send(aborted) to c
    else
      TO-broadcast(⟨readSet_t, writeSet_t, c, s_d⟩)
      to S
    end if
  end when
end
```

WEAK VOTING REPLICATION

- Client, C sends a transaction, t to server, S_d
- S_d executes t but delays write operations



- When commit time is reached, the delayed write set in t is put into a Message, m and broadcasted to all servers using total order
- Upon delivering m, the delegate server, S_d determines if the transaction, t can be committed or not
- Based on the determination, S_d sends a second broadcast with Abort or commit decision

Delegate Server, Sd

```
task Processing
  {Executed by delegate server s_d}
  when receive transaction t from
  client c
    execute transaction t
    if aborted(t) then
      send(aborted) to c
    else
      TO-broadcast(⟨writeSet_t, c, s⟩) to S
    end if
  end when
end
```

Any Server, Si

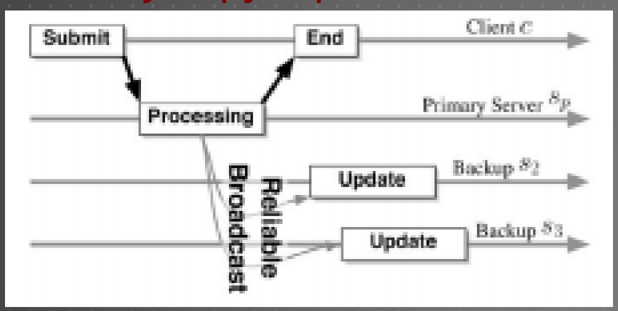
```
task Execution
  {Executed by server s_i}
  when
  ⟨writeSet_t, c, s_d⟩
    if s_d = s_i then TO-deliver
      status_t ← vote(t)
      R-broadcast(status_t) to S
      send(status_t) to c
    else
      execute writeOperations_t
    end if
  end when
end
```

```
task Termination
  {Executed by server s_i}
  when R-deliver(status_t)
    if status_t = commit then
      commit(t)
    else
      abort(t)
    end if
  end when
end
```

PRIMARY COPY REPLICATION

- All transactions from any Client, c are sent to one server, S_p
- No other server accepts transactions from any client
- All other servers serve as backups
- The serialization order and abort or commit decisions are made by S_p
- The transaction is processed at S_p and updates are sent to all other servers using a reliable broadcast

Primary copy replication scheme



Primary Server, Sp

```
task Processing
  {Executed by primary}
  if primary(s) then
    when receive transaction t from client c
      process transaction t
      status_t ← try-commit(t)
      if status_t = committed then
        update_t ← updates done by t
        R-broadcast(update_t) to S \ s
      end if
      send(status_t) to c
    end when
  end if
end
```

Backup Server, !Sp

```
task Update
  {Executed by backup}
  if ¬ primary(s) then
    when R-deliver update_t
      process(update_t)
    end when
  end if
end
```

LAZY REPLICATION (FOR COMPARISONS ONLY)

- A Client, C sends a transaction, t to a server, S_d
- S_d executes t and send updates are broadcasted to others servers

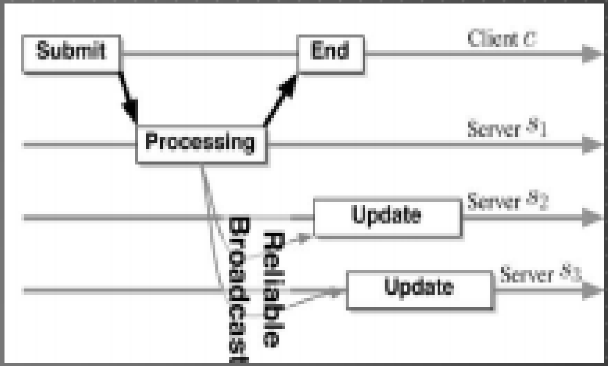
Delegate Server, Sd

```
task Processing
  {Executed by all servers}
  when receive transaction t from client c
    process transaction t
    status_t ← try-commit(t)
    if status_t = committed then
      update_t ← updates done by t
      R-broadcast(update_t) to S \ s
    end if
    send(status_t) to c
  end when
end
```

```
task Update
  {Executed by all servers}
  when R-deliver update_t
    process(update_t)
  end when
end
```

All other servers

Lazy Replication Scheme



EXPERIMENTS

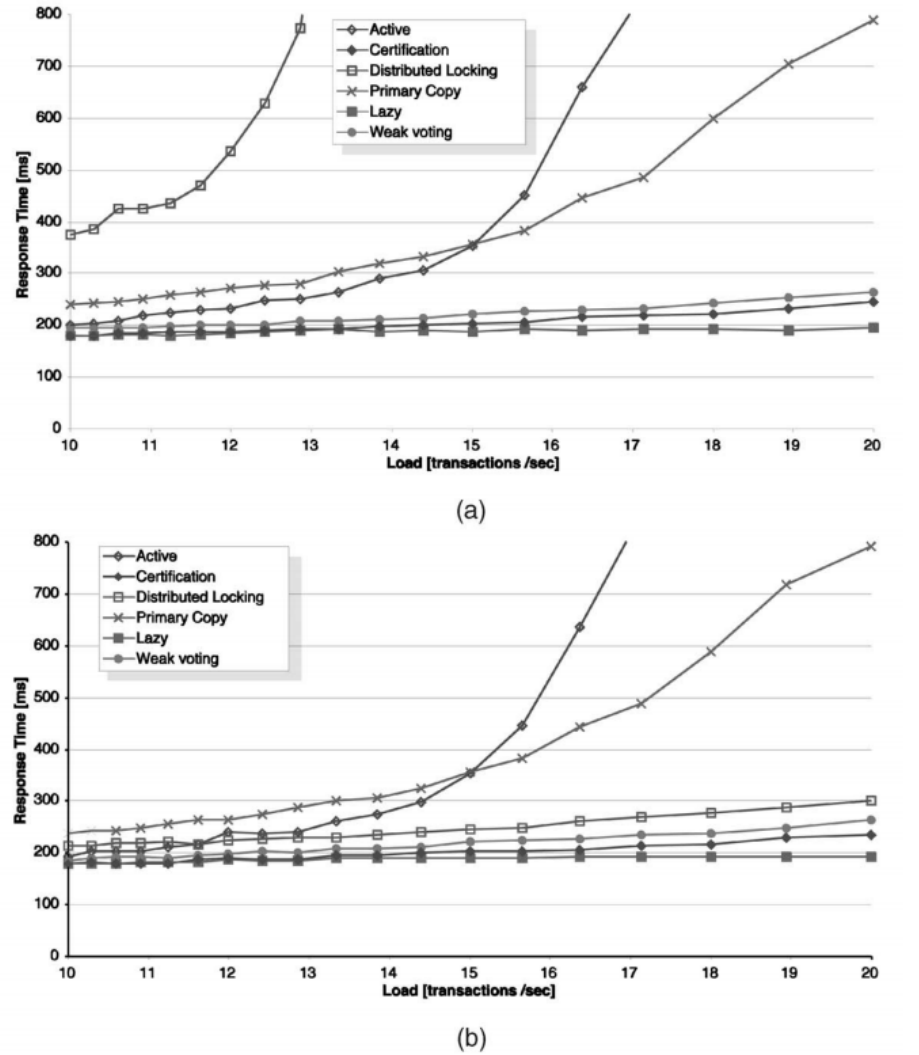


Fig. 11. Overall performance medium-load (a) slow network and (b) fast network.

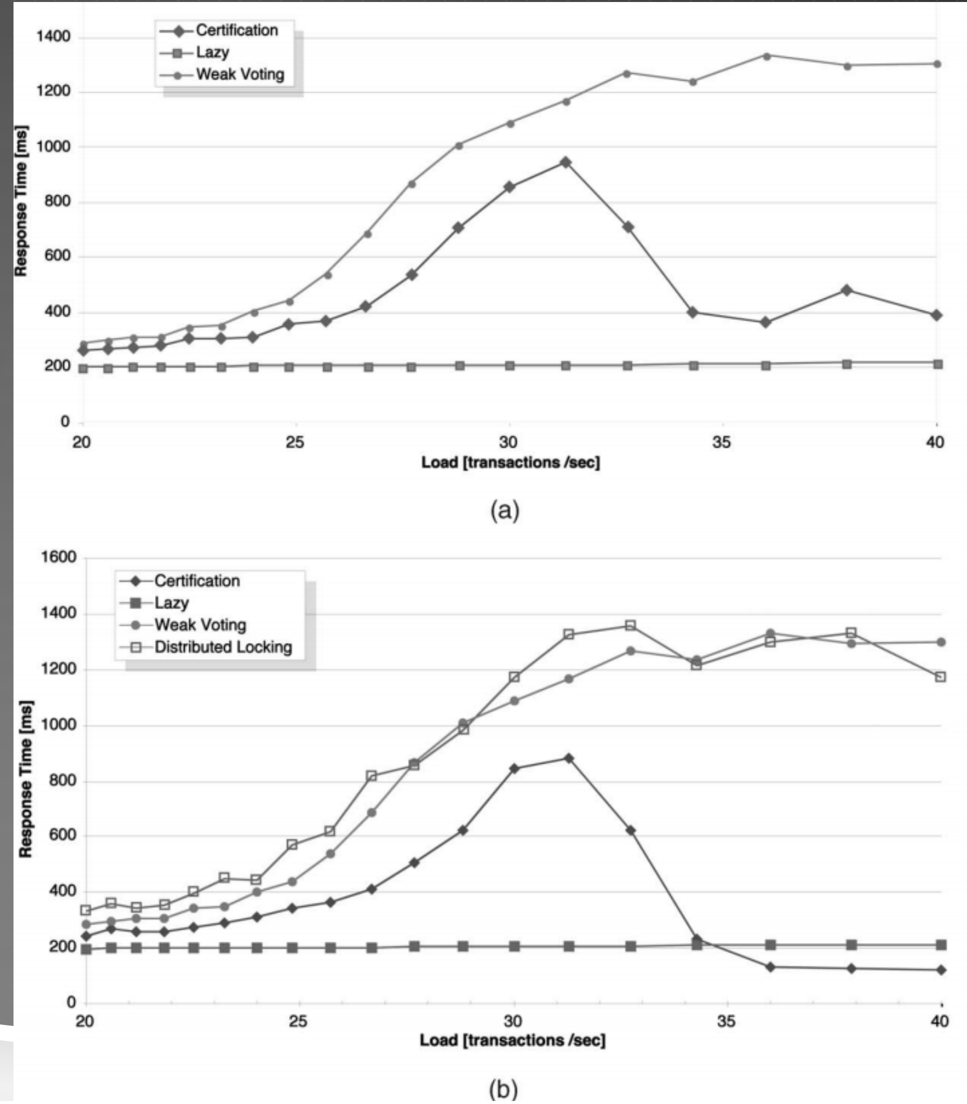


Fig. 13. Overall performance high-load (a) slow network and (b) fast network.

EXPERIMENTS CONT'D

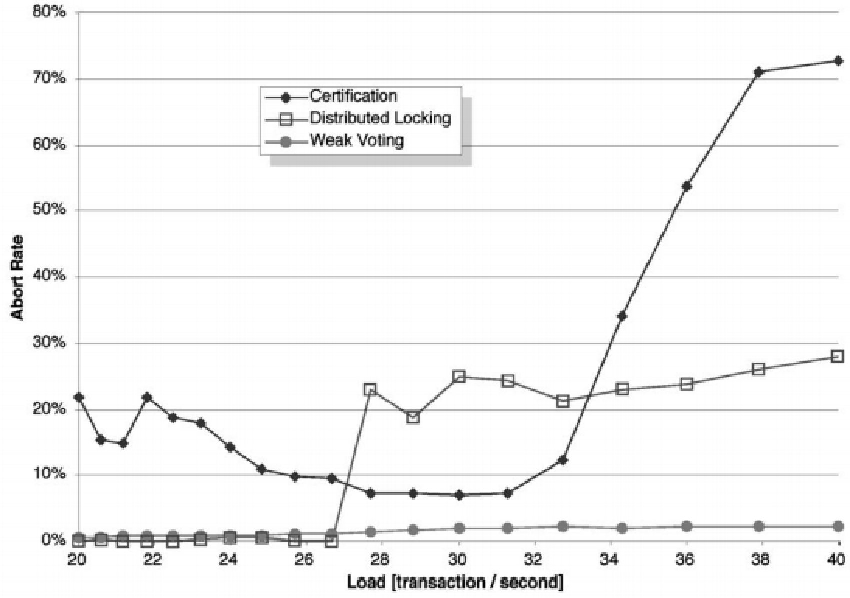
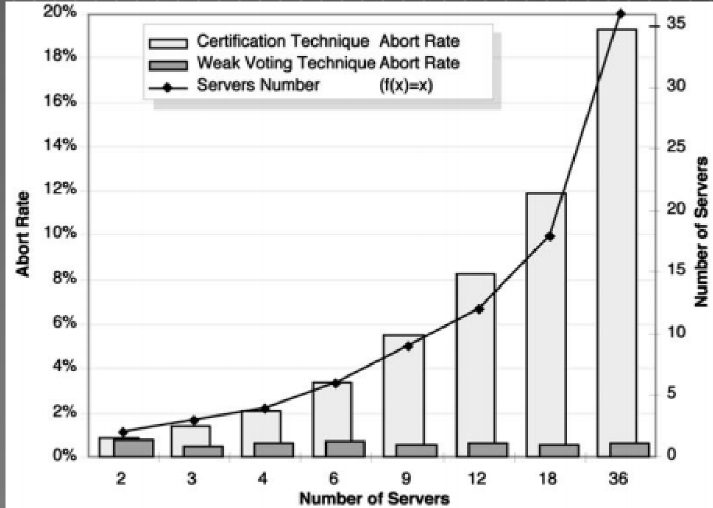


Fig. 14. Abort rate with high-load, fast network.



(a)

Fig. 16. Abort rates as a function of (a) the number of servers and (b) the load of the system.

EXPERIMENTS - SCALABILITY

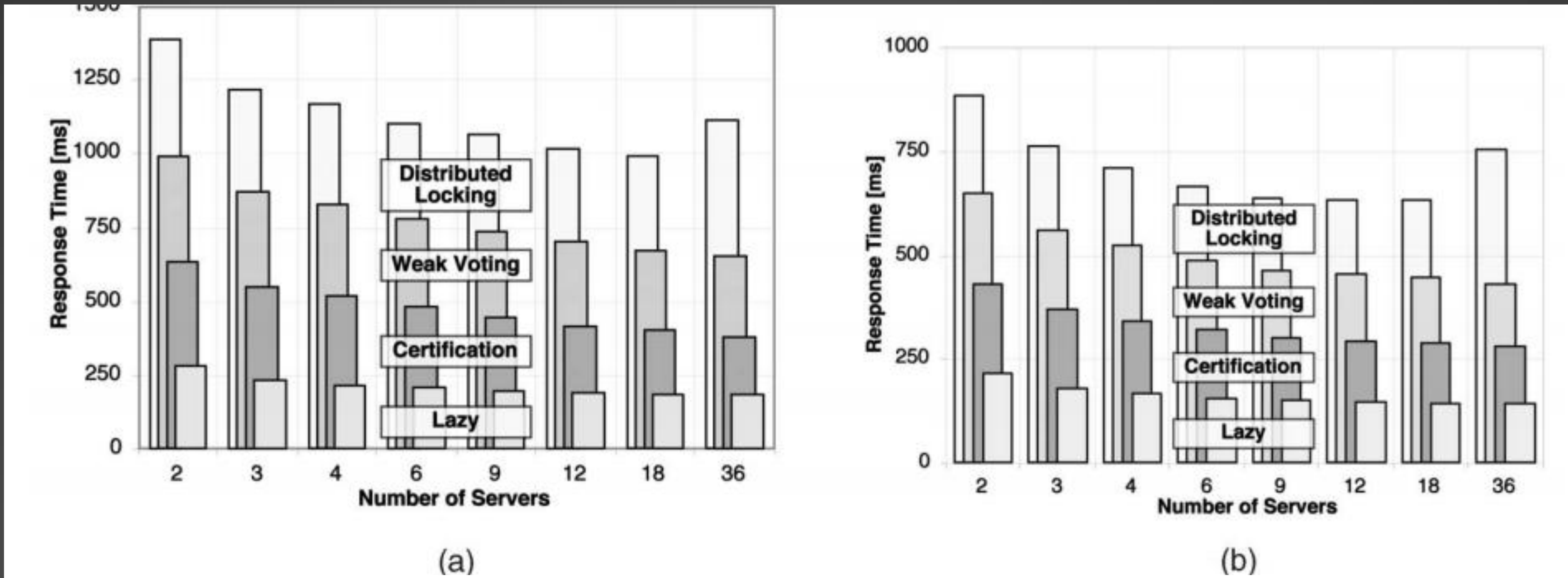


Fig. 15. Scalability with (a) a query rate of 50 percent and (b) a query rate of 80 percent.

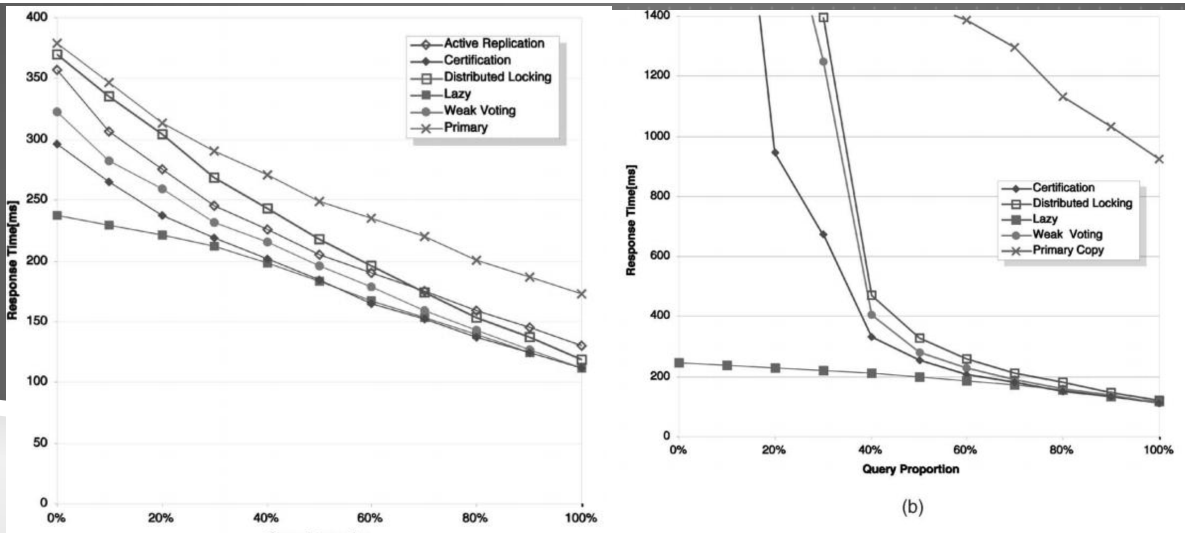


Fig. 18. Performance with changing query rate at (a) 10 transactions per second, (b) 20 transactions per second.

ZOOKEEPER: WAIT-FREE COORDINATION FOR INTERNET-SCALE SYSTEMS

HUNT, KONAR, JUNQUEIRA, AND REED

INTRO

- Provides coordination framework for large-scale distributed applications
- Manipulation of data objects that are organized hierarchically resembling a file system structure
- Guarantees FIFO ordering for all operations
- Leader based atomic protocol ;Zab
- Writes are linearizable
- Allows local data caches that are managed by clients
- Utilizes a watch mechanism; A client watches for an update to a given data object and receives notification upon change

ZOOKEEPER SERVICE

- Znodes; Abstraction of a set of data nodes organized according to hierarchically namespace
- Znodes
  - Regular
    - Explicit deletion
  - Ephemeral
    - Explicit of automatically

    deleted by the system
  - Can be created by setting a sequential flag
    - When a new node is created with this flag, a monotonically increasing counter is appended to the node's name
      - The number attached to the name is never higher than a preexisting sibling's number
- A watch flag can be set during a read operation
  - When it is set
    - A client receives a one time notification about a change of that data object

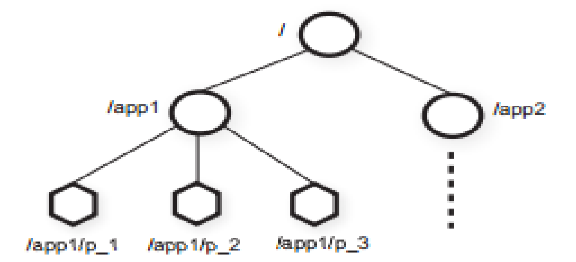


Figure 1: Illustration of ZooKeeper hierarchical name space.

- Data Model
  - A non general purpose file system with simplified API
  - Full data reads/writes
- Sessions
  - Initiated by connecting to Zookeeper
  - Terminated
    - When Zookeeper does not receive *word* for more a set time (timeout)
    - A client explicitly closing a session
    - A client is deleted because it is faulty
  - Enables clients to persists across servers

SOME IMPORTANT CLIENT API

- create(path, data, flags)
  - Creates a znode with path name path, stores data[] in it
  - returns the name of the new znode
  - flags enables a client to select the type of znode: regular, ephemeral, and set the sequential flag;
- delete(path, version):
  - Deletes the znode with the path if that znode is at the expected version
- exists(path, watch)
  - Returns true if the znode with path name path exists, and returns false otherwise. The watch flag enables a client to set a watch on the znode
- getData(path, watch)
  - Returns the data and meta-data, such as version information, associated with the znode.
  - The watch flag works in the same way as it does for exists(), except that ZooKeeper does not set the watch if the znode does not exist;
- sync(path)
  - Waits for all updates pending at the start of the operation to propagate to the server that the client is connected to.
- All methods have both asynchronous and synchronous versions

PRIMITIVES

- Configuration Management
- Rendezvous
- Group Membership
- Simple Locks
- Simple Locks without Herd Effect
- Read/Write Locks
- Double Barrier

Configuration Management (dynamic configuration)

- Imagine a regular non distributed application
- Imagine the application have an updatable ‘config ‘ file that the app reads from at some time in the life of that app

- Now, imagine implementing this with Zookeeper
  - System configuration is stored at znode Zc
  - Each process starts by knowing the path to Zc
  - Each starting process obtains its configuration by reading Zc and setting the watch flag
  - When Zc changes, the processes are notified
  - They reread Zc and set the watch flag again

Rendezvous

- When a final system configuration cannot be determined at the beginning of a system but unavailable information about a subset of the system has to be passed to some subset of the system, Zookeeper can utilizes its watch feature to solve this problem.
  - For example, a client may want to start a master process and several worker processes, but the starting processes is done by a scheduler, so the client does not know ahead of time information such as addresses and ports that it can give the worker processes to connect to the master.
- Let Zd be designated znode.
- At the start of the system, the processes interested in the information {pi} are given the path to Zd
- {pi} read Zd and set a watch flag
- When the information is known, Zd is updated and {pi} is notified.
- {pi} rereads Zd and set watch flag again and cycles continues

Group Membership

- Recall that ephemeral znodes are just like normal znode but can be removed automatically when the node fails
- Group membership can be implemented using Zookeeper
  - Let Zg be a designated znode that represents a group, g
  - Any znode created as child node to Zg is in group, g
  - Finding out information about group, g is as simple as reading the children of g
  - In order to have unique children of Zg, unique names can be given or the sequential flag can be set when creating an ephemeral znode
  - Any process, pi that wishes to monitor changes in group, g, can set a watch flag to Zg and be notified when ever there is a change in that group
  - Pi can then read Zg and set the watch flag to true and repeat
  - Since ephemeral znodes are sort self maintaining, when a child znodes to Zg dies, group membership is automatically modified to reflect the new state

SYSTEM PERFORMANCE

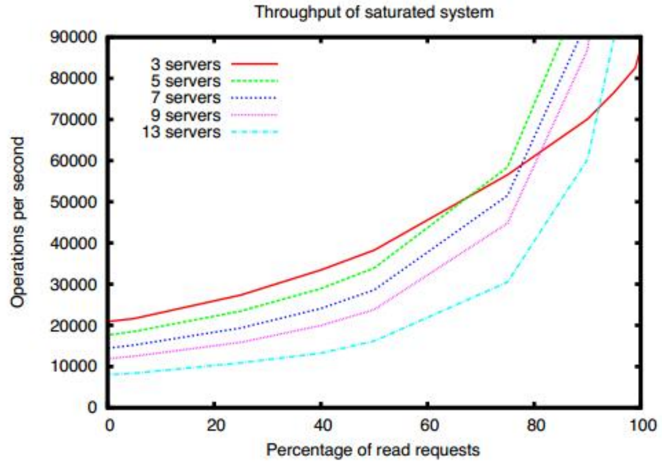


Figure 5: The throughput performance of a saturated system as the ratio of reads to writes vary.

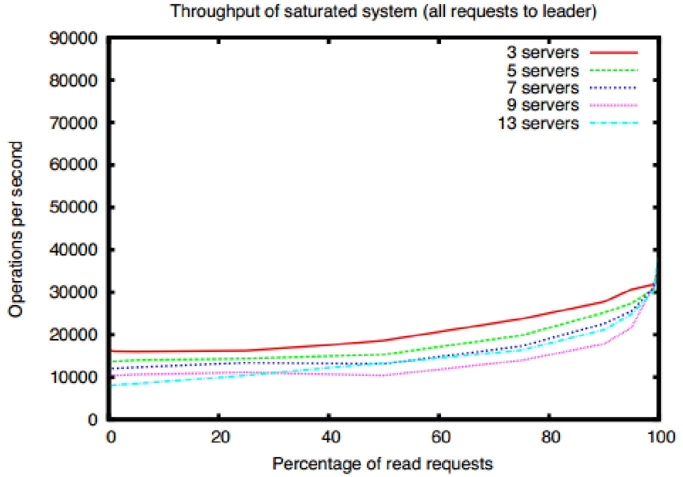


Figure 6: Throughput of a saturated system, varying the ratio of reads to writes when all clients connect to the leader.